数理解析研究所講究録
461 巻 1982 年 151-172

A BRIEF TUTORIAL ON THE PHYSICAL REPRESENTATION OF DATABASE STRUCTURE

Isamu KOBAYASHI

SANNO Institute of Business Administration
School of Management and Informatics
Kamikasuya 1573, Isehara, Kanagawa 259-11

ABSTRACT: A systematic discussion of the physical representation of database structure is evolved. Design of physical database organization is composed of two techniques; representation of each unit datum by an appropriate bit string, and formation of multilevel associations among unit data each represented by a bit string. Various file organizations are discussed first, then representation of entity relations and that of relationship relations are discussed separately.

Keyword and Phrases: database, database organization, entity relation, file organization, physical representation, relationship relation

CR Categories: 4.33, 4.34

INTRODUCTION

After the logical database structure has been clearly defined, information sytems designers must determine physical representation of this logical database structure. This is particularly important when a database management system is to be designed to support information systems construction. There are many excellent materials [Knuth 1975, Martin 1976, Wiederhold 1977] describing physical file or database organizations; however, all of them are too thick to get an overview easily. Also, many materials use several ill-defined terms (for example, ISAM and VSAM, which are actually not access methods but file organization methods). These makes it very difficult for the systems designers to select the best representation for their applications.

In this paper, the author tries to present a brief but still systematic discussion of the physical representation of data structure. The paper does not include much quantitative discussion. However, it may become a good guide for the first step of physical database design procedure.

UNIT DATA REPRESENTATION AND ASSOCIATION

Physical representation of database structure is basically composed of two techniques; expression of individual values by appropriate bit strings stored in computer storage, and formation of various multi-level associations among these bit strings. The former is almost application-dependent. If the value is like the scalar or subrange type in PASCAL, its representation is very easy. If the value is like the structured type, we must devise a more complicated expression. We would not discuss this problem because there may be an infinite number of data types to be represented.

The organization of memory in human brains is not well understood. We can suppose, however, that human brains use many types of multi-level associations among data elements. There seens to be associations of various strength. Some seems very strong, while some others very weak.

In contrast, fewer methods of association are available in computer storage. The following four are basic. The first method is association by arranging individual values represented by bit strings

adjacently on physical storage devices. One or more values can be associated in this way. The values associated in this way are collectively called a (physical) *record*, in which each value is called a *field*.

The second method associates a set of records (composed of one orr more values) by allocating them in computer storage using a common address assignment algorithm. As mentioned later there are various algorithms for assigning storage addresses to records. Such a set of records are collectively called a (physical) *file*.

The third method associates several records by giving them a common value in a specific field called a *linking field*. (A foreign key can be represented by a linking field.)

In the fourth method, a linking field is replaced by a *pointer field*, which is the physical (absolute or relative) address of another record to be associated with the record containing this pointer field. Pointer organization can be bi-directional, that is, two records can have pointers pointing to each other. Several different pointer organizations are possible when this method is used for associating more than two records, some of which are illustrated in Fig. 1.

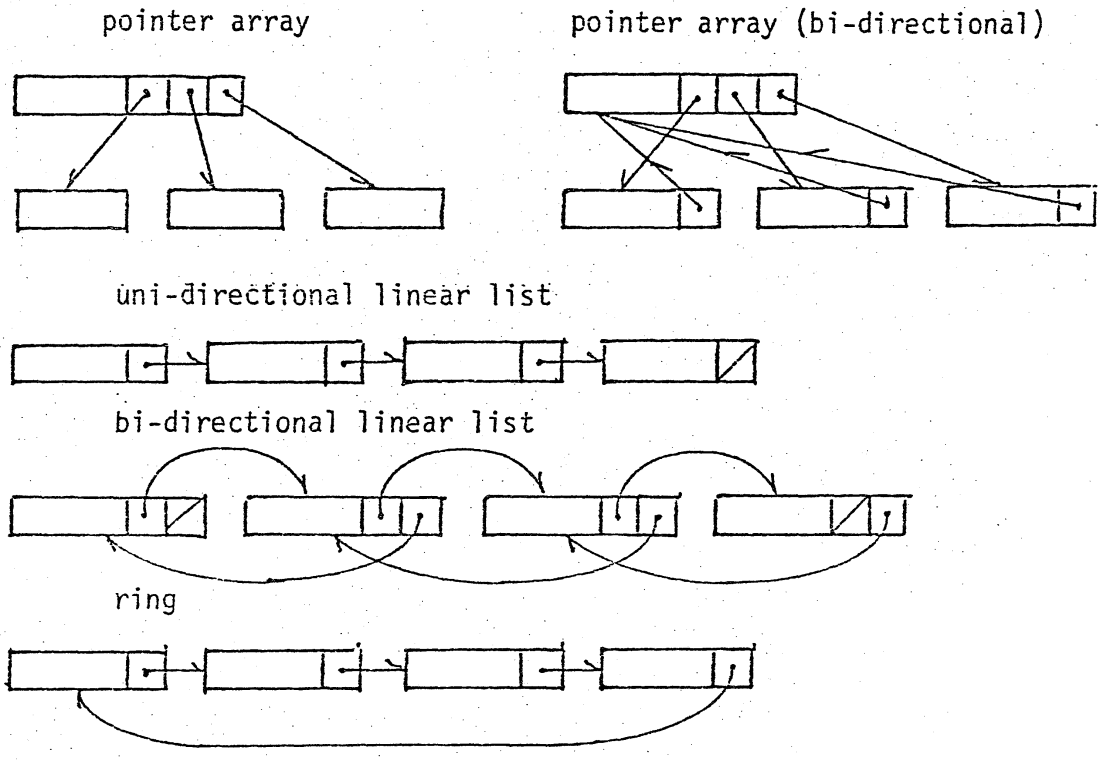

Fig.1 Basic Pointer Organizations

The logical database structure established through the logical design phase of information systems must be mapped into a physical database that is structured using these associations.

BASIC FILE ORGANIZATIONS

First we will discuss several different address assignment algorithms for organizaing a file.

(1) Heap File Organization

Records can be stored in a contiguous storage areas in the order of their arrival. Such a simply organized file is called a *heap file*. If a record is deleted from this file, the location having been occupied by the deleted record can be assigned to a new record generated afterwards. To achieve such a reassignment, an appropriate file area control program must be provided, which monitors the areas occupied by the existing records as well as the areas available for newly added records.

The heap file organization can be used to store either fixed-length or variable-length records. If it is used for the latter, a number of small alots that cannot be used to store newly added records may be generated after repeated updates of the file. A file reorganization called *garbage collection* should be performed at appropriate times to rearrange records in the file and to eliminate such unusable slots.

(2) Sequential File Organization

Records can be stored on a contiguous storage area in the order of a certain field value. The field used to arrange records in this file is called the *key field*. The primary key attribute of tuples (logical records) is not always necessarily but usually represented by the key field. A file organized in this way is called a *sequential file*.

The sequential file organization can be used to store either fixed-length or variable length records. This file organization is mandatory on sequential storage devices. It can also be applied to direct access storage devices; however, the update efficiency becomes very poor because addition and deletion of records always require rewriting a large part of the file.

(3) Partitioned Sequential File Organization

In order to improve the update efficiency of a sequential file placed

in direct access storage devices, it can be divided into partitions of an appropriate size, each containing several records. This differes from the sequential file, in which addition or deletion of a record requires rewriting all the records located after the added or deleted record. Update is propagated only within a partition in the partitioned sequential file. If an overflow occurs in a partition, this partition is divided into two partitions. Conversely if all records in a partition are deleted, this partition itself is removed from the file. Prtitiones are assigned addresses in a similar way to the heap file organization. Each partition in a file contains a pointer to the next partition (and a pointer to the previous partition). In this sense, the partitioned sequential file organization uses both the second and fourth methods of association.

(4) Tree-structured File Organization

As well known, we can use an n-ary search algorithm on a sequential (and partitioned sequential) file to improve a certain kind of search operations. This n-ary serach algorithm can be embodied in a *tree-structured file* organization. A tree-structured file is composed of a number of partitions each containing n-1 records and n pointers. Let $K(r)$ be the key field value of the record r. Records $r_1, r_2, \ldots, r_{n-1}$ in a partition are arranged in the sequence of their key field values, that is,

$$K(r_1)<K(r_2)<\ldots<K(r_{n-1}).$$

The pointer p_1 points to a partition containing records all whose key field values are smaller than $K(r_1)$, the pointer p_k $(2\leq k\leq n-1)$ points to a partition containing records all whose key values are between $K(r_{k-1})$ and $K(r_k)$, and pointer p_n points to a partition containing records all whose key field values are greated than $K(r_{n-1})$. Fig. 2 shows a ternary tree-structured file.

To achieve better search efficiency, all branches of each partition must contain an almost equal number of partitions. Such a tree is called a *balanced tree*.

A tree-structured file can be maintained in a similar manner to maintaining a partitioned sequential file. However, to obtain a balanced tree, a dynamic file reorganization algorithm can be integrated. The *B-tree file* organization is a tree-structured file

organization in which a means to always produce a balanced tree is integrated.

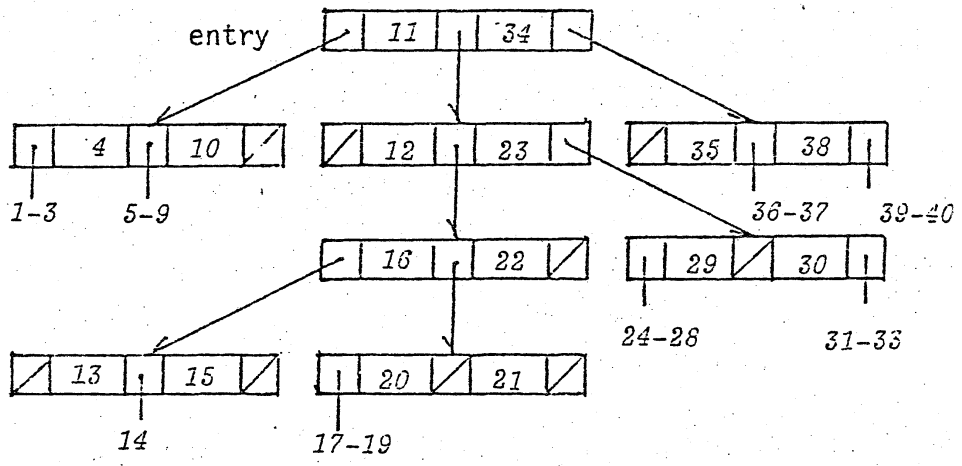

Fig.2 A Tree-structured File

(5) Direct File Organization

The *direct file* organization uses the key field in allocating records in a file but in a different way. This file organization is achieved for a file F by assigning the address calculated by

$$A(r)=\ell\times N(K(r))+b$$

to each record in it, where ℓ is the record length that must be fixed for the file, N is an integer-valued function, and b is a constant that specifies the beginning address of tne area assigned to this file.

The function N must be an injection (one-to-one mapping from the set of key field values into a certain set of integers). In addition, the cadinality of I-I', where

$$I=\{i \mid \min_{r\in F} N(K(r)) \leq i \leq \max_{r\in F} N(K(r))\}$$

and

$$I'=\{N(K(r)) \mid r\in F\}$$

must be as small as possible. In general, it is very difficult to find such an integer-valued function.

One solustion is to neglect the first restriction. We may provide a *bucket* that can accommodate several (or just one) records *colliding* with each other (assigned the same address). If a bucket overflow occurs, the record causing the overflow is assigned an open (not occupied by another record) address in the next bucket or in an overflow bucket provided in some separate area (organized as a heap file) linked to the overflowed bucket by a pointer. To make the number of collisions

as small as possible, a certain *hashing function* is usually used for the function *N*.

The five basic file organizations mentioned above are mutually *exclusive* except the binary tree-structured file organization, that is, no more than one distinct file organization can be applied to organizaing a file. The binary tree-structure file organization is non-exclusive because every partition in a binary tree-structured file contains only one record and each record can be assigned an address without regard to the addresses assigned to any other records. Therefore, we can overlay a binary tree-structured file organization on whatever file organization we have employed.

Among the five basic file organizations, the heap, and direct file organization are *static* in the sense that the address once assigned to a record is never changed unless a certain file reorganization such as garbage collection is invoked. In contrast, the sequential, and partitioned sequential file organization is *dynamic* in the sense that the address assigned to a record can be altered whenever an update (adding or deleting a record) is applied to the file. We know that the sequential file is more dynamic than the partitioned sequential file. The tree-structured file organization is basically static. However, if a dynamic file reorganization procedure is integrated to obtain a balanced tree for an n-ary ($n \geq 2$) tree-structured file like a B-tree, the file organization becomes dynamic.

REPRESENTATION OF ENTITY RELATIONS

In many data models, entity relations are distinguished from relationship relations. This distinction can be formally descibed as difference of semantic constraints holding in both types of relations [Kobayashi 1981a]. It is particularly significant when we consider navigations along relationship relations and represent them differently from entity relations.

A tuple in an entity relation may be represented by one record, or by several records linked together by linking fields or pointers. The latter representation is desirable when the tuple has several groups of attribute values to be accessed in considerably different

frequencies, or when the tuple has both fixed-length and varuable-length attribute values. In these cases each group of attribute values may be represented by one record. One of the records that together represent a tuple is called the *main record* (in most cases, composed of fixed-length attribute values), while others are called the *subordinate records*.

An attribute value is represented by a field in a record. If the attribute value is compound, the corresponding field is composed of a fixed or variable number of *subfield*.

Main records representing tuples in an entity relation are organized to form a file using one of basic file organizations. Subordinate records of the same type (format), if exist, compose another file. There may several different types of subordinate records, and hence several files each composed of subordinate records of a certain type may be created. These files are also organized using one of basic file organizations. A *main file* consisting of main records and *subordinate files* consisting of subordinate records together represent an entity relation.

Physical representation of an entity relation is not uniquely determined. In fact, decomposition of a tuple into one or more parts each represented by a record, selection of the linking method for associating a main record and one or more subordinate records, selection of the file organization for organizaing the main and subordinate files, are all not uniquely determined. We must select a specific representation from a variety of possible representations with various qualitative and quantitative factors taken into consideration. Next we will discuss how each file organization behaves when basic database operations are applied to it.

(1) Exhaustive Search and Sequential Search

We will first examine various search operations applied to a file. As a special case the search condition can be $T(r)$ which assigns all the records in the file the value 'true'. The search with $T(r)$ is called an *exhaustive search*. If an exhaustive search must be performed in the order of the key field value, it is called a *sequential search*.

A heap file is efficient for the exhaustive search but is inadequate for the sequential search. A sequential file is efficient for the sequential

search because it can be achieved by physical sequential read operations. The partitioned sequential file organization is less efficient than the sequential file organization but still fairly efficient for the sequential search. The tree-structured file organization is not so efficient as these two for the sequential search although the sequential search can still be performed on it. The direct file organization is quite inadequate for any exhaustive search. It is almost impossible to perform a sequential search on it.

Although we do not use here, the sequential search is in many cases called the sequential access method (SAM).

(2) Equal Key Search

The *equal key search* is another special case of search operations, for which the search condition is of the form

$K(r)$=const.

To perform an equal key search (as well as other more complicated search operations described later) on a heap file, a *seek*, which fetches all records in the file one by one and tests them against the given condition, must be carried out.

This time-consuming procedre can be somewhat improved when searching a sequential file using a well-known n-ary search procedure. The search time required for the seek is $O(n)$, where n is the number of records in the file, while that required for the n-ary search becomes $O(\log n)$. The n-ary search is not easily achieved on a partitioned sequential file because partitions in it are linked together by pointers. On the other hand, the n-ary search can be very efficiently achieved on an n-ary tree-structured file because the n-ary search is embedded in the associations among partitions.

The equal key serach is very efficeintly performed on a direct file The equal key search applied to a direct file is called the direct access method (DAM).

(3) Key Search

The *key search* is a search operation with a search condition of the form

$K(r)\ \theta$ const

where θ is an arbitrary relational operator other than =. The n-ary search procedure is still applicable to a sequential file, a partitioned sequentail file and an n-ary tree-structured file as well. In

contrast, no efficiant search procedures are available for a direct file to achive any key search, or it is even impossible to achiev a key search on it.

(4) Non-key Search

The *non-key search* is a search operation with a search condition of the form

$A(r)\ \theta$ const

where $A(r)$ is the value of an arbitrary field of record r, and θ is an arbitrary relational operator (θ can be =). The value $A(r)$ can be replaced by $f(A(r))$ where f is an arbitrary function of $A(r)$.

If $A(r)$ is not $K(r)$, all the basic five file organizations are powerless in improving the efficiency of the non-key search, and hence a seek must be carried out (except on a direct file, for which the seek operation itself is very difficult to operate).

One way to shorten the seek time is to let the record size as small as possible. To deal with the non-key search, we may use the record that has only two fields; one is $A(r)$ and the other is a linking field or a pointer field used for associating it with the record r. This small record is called the *index record* of r regarding A. Index records of records in a file are organized to form an *index file* using the sequential, partitioned sequential or tree-structured file organization with $A(r)$ being the key field of the index file to enable an efficient sequantial or n-ary search.

The non-key search can be performed by a search on the index file regarding the field A (by a sequential or n-ary search) followed by fetching the associated record. The latter step can be achieved by an equal key search if a linking field containing $K(r)$ is used, or by traversing the pointer if a pointer field is used. This procedure may be called the indexed access method (IAM). If the file to be indexed is formed by a dynamic file organization, the linking field association is mandatory. If it is formed by a static file organization, the pointer field association, which is more efficient than the linking field association in finding the associated record, can be employed. Index files are not necessarily indexed; therefore, they can be formed by a dynamic file organization. The B-tree organization is widely used.

If the field regarding which an index file is to be created is not the key field, more than one index records with the same $A(r)$ value (the key field of the index file) can be generated. These records can be stored

in contiguous positions in the index file. They can also be collected into a single index record whose linking field (or pointer field) is an array of key field values of associated records (or pointers pointing to associated records). A pointer array can be replaced by a linear list or a ring pointer organization. Such a method of forming association is called a *multilist* or *multiring*. The mulyilist (and multiring) organization is good for processing a single non-key search but it is not advisable when the non-key search must be performed as a unit search in a compound search described later.

Index files can be created so that each index record points to the partition or bucket to which the associated record belongs. Also for a partitioned sequential file only the first record of each partition can be indexed. This organization is called the *indexed sequential file* organization. Both the sequential search (SAM) and search using the index file (IAM) are possible on a indexed sequential file. (No search operations corresponding to the term ISAM exists!)

If $A(r)$ is an fixed or variable size array, a search with a search condition of the form

$\text{const} \epsilon A(r)$

may become necessary. In this case, the index file can be modified so that as many index records as the number of components of $A(r)$ value each having a component value as its key field and pointing to the same record are generated for a single record. Such index files are very desirable in, for example, document retrieval applications.

Like the binary tree-structured file organization, the index file organization is non-exclusive. Therefore, indexs files can be created for any number of fields in the record. Index files can be created even for values attached to but not explicitly represented as fields in the record. These index files can be created regardless of what file organization is employed for the file to be indexed. However, the presence of index files degrades the update performance because they must be updated whenever the indexed file is updated. We must be very careful in selecting fields for which index files are to be created [King 1974, Kollias 1979, Lum 1971, Palermo 1970, Senko 1973, Schkolnick 1975].

Index files for two fields $A_1(r)$ and $A_2(r)$ can be used to improve the search with a search condition of the form

$f_1(A_1(r)) = f_2(A_2(r))$

where both f_1 and f_2 are constantly increasing (or decreasing) functions. two index files provided for A_1 and A_2 are sequentially collated and, for each match, it is examined whether the two have common linking field values or pointer values. Such common values, if exist, determine the records qualified for the given search condition.

(5) Compound Search on a File

There may be various other unit searches to be applied to a single file for which no efficient search procedures are available. They as well as non-key searches for which index files prepared for the file on which the search must be performed cannot be used must be achieved by a seek.

In general, a search on a single file may be requested with a compound condition composed of several unit conditions combined by logical operators $\vee$ and/or $\wedge$. (The $\sim$ operator can be eliminated by changing a relational operator in the unit condition appropriately.)

A search with two unit conditions λ_1 and λ_2 combined by an $\vee$ or $\wedge$ operator can be processed collectively by a seek. However, as we have

$$s[\lambda_1 \vee \lambda_2](F) = s[\lambda_1](F) \cup s[\lambda_2](F)$$

and

$$s[\lambda_1 \wedge \lambda_2](F) = s[\lambda_1](F) \cap s[\lambda_2](F),$$

where $s[\lambda](F)$ is the result of the search on F with the search condition λ, we can use two unit searches followed by a union or an intersection operation to obtain the same result. If an efficient search procedure is availabe to achive both unit searches and if the search results contain a fairly small number of records, the latter procedure is faster than the seek. The unon and intersection operations can be achieved efficiently by sorting both search results and then collating them sequentially.

Also as we have

$$s[\lambda_1 \wedge \lambda_2](F) = s[\lambda_1](s[\lambda_2](F)) = s[\lambda_2](s[\lambda_1](F)),$$

we can process $\lambda_1 \wedge \lambda_2$ by a search on F with λ_2 (or λ_1) followed by a search on its result with λ_1 (or λ_2). If the former search can be performed efficiently and if the search result contsins a fairly small number of records, this procedure is much faster than the seek, even if the latter search must be performed by a seek.

Given a compound search condition, we can make a syntactical

transformation to maximize the use of efficeint search procedures supported by the file organization and index files. This problem has been studied by several reserachers [Astrahan 1975, Kobayashi 1976].

(6) Search on more than one File

Generally, the search

$s[\lambda](F_1, F_2, \ldots, F_m)$

is performed with a logical function λ defined on $F_1 \times F_2 \times \ldots \times F_m$. The function λ may have one or more variables bound by existential or universal quanitiers or by some other means. The matrix part of its prenex normal form is composed of one or more unit conditions combined by logical operators. For some unit conditions defined on a single file, we already have some means of improving the unit search efficiency.

If a unit condition itself is defined on a Cartesian product of more than one file, then we should perform a seek on this Cartesian product. The seek time on $F_1 \times F_2 \times \ldots \times F_\ell$ is $O(n_1 \times n_2 \times \ldots \times n_\ell)$, where n_k is the number of records in F_k.

However, for a component condition of the form

$$\bigwedge_{k=1}^{\ell} (A_1(r_1) = A_k(r_k)) \quad (\text{for } r_k \in F_k)$$

or that of some equivalent form, a sequentail collation of ℓ files $F_1, F_2, \ldots, F_\ell$ can be used. The time required for this procedure is $O(n_1+n_2+\ldots+n_\ell)$, To achieve the sequential collation, files to be collated must have been sorted in the sequence of A_k value. If a sort operation is necessary (if F_k is not sequentailly organizaed with A_k as its key field), we need $O(n_k \log n_k)$ as the sort time. In the worst case, the time required to process this condition becomes $O(\Sigma_{k=1}^{\ell} n_k \log n_k) + O(\Sigma_{k=1}^{\ell} n_k)$, which is still much less than the time required for the seek in most cases. If an index file is provided for A_k, it can be used in the sequential collation instead of the file on which the search is to be performed.

In general, there can be more than one logical function logically equivalent to each other. Given a compound search condition defined on more than one file, we can make a syntactical transformation to obtain a logical function, which is equivaled to the given search condition but maximizes the use of sequential collations as well as index files. Component conditions for which an efficient search procedure is available nust be processed before processing

other component conditions. This problem has been studied also by several researchers. Some studied the optimal strategy for searches on two files [Smith 1975, Yao 1979]. For a class of logical functions defined as relational calculi, Codd [1972] presented a decomposition of the search operation into relational algebra operations but no optimization issues were included. This work has been extended by several researchers [Palermo 1972, Rothnie 1975, Wong 1976, Held 1975] with an optimization issue taken into consideration. To process quantifications, projection and division in the relational algebra were used. For a larger class of logical functions defined as extended relational calculi, Kobayashi [1981c] presented a general search strategy, in which the search operation was decomposed into extended relational algebra operations.

(7) Update

The update efficiency depends on how a unit update propagates other unit updates. If the file organization is static, no propagation is occurs. Therefore, the heap and direct file organization are very efficient for updates. Also, the basic (in the sense no dynamic reorganization is integrated) tree-structured file organization is efficient for updates. On the other hand, the sequential file organization is quite inefficient for updates. Both the partitioned sequential file organization and the tree-structured file organization in which a dynamic reorganization procedure is integrated show an intermediate efficiency for updates.

As mentioned previously, provision of index files considerably degrades the update efficiency. One or more unit updates on the index files are usually propagated from a unit update on the indexed file.

We will not go further into quanyitative discussions of selecting file organizations to represent entity relations because further discussions must depend on specific hardware devices and specific implementation on them. There are several quantitative discussions found in materials like Knuth [1968], Martin [1976] and Yao [1976].

REPRESENTATION OF RELATIONSHIP RELATIONS

Unlike entity relations, tuples in relationship relations are not

always represented directly by (physical) records. In particular, when the relationship relation to be represented is binary and has no more attributes than those pointing to its origin and destination, it is usually represented quite differently from representing an entity relation. We will next discuss these different representations and the efficiency of navigations along the relationship relation and updates on it.

(1) Sequential Record Arrangement

A tuple in a binary relationship relation is a relationship between two tuples in one (if the relationship relation is defined on a relation) or two (if it is defined between two relations) relations. Such a relationship can be represented by forming an association between the two records representing the two tuples to be related by it. The first, third and fourth methods of association described previously can be used to form this association.

The first method, which is to store the two records adjacently, can be used only when the relationship relation is a sequence or a quasisequence (a union of several sequences, which are disjoint with each other). If the relationship relation is not a sequence, an explicit indication spcifying apices and/or terminal records with respect to the quasisequence must be provided (by adding an additional field or some special records). The obtained file resembles a sequential file mentioned previously but no key field are used in organizaing it. Note that this organization is still exclusive in organizing a file for the relation on which the relationship relation is defined.

This representation has two advantages. One is that it does not use any additional storage spaces to represent the relationship relation. The other is that navigations along this relationship relation can be very efficiently achived by physical sequential read (forward or backward) operations. On the other hand, the update efficeincy is very poor because either ap update of the relation on which the relationship relation is defined or an update of the relationship relation propagates many other updates.

(2) Linking Field Representation

The third method of association using a linking field can also be used to associate two records representing two tuples related by a

relationship. Usually a foreign key is represented by the linking field.

In some cases, a special field expressing the relationship directly can replace the linking field. For example, a set of ordinal numbers can represent a sequence and a set of Dewey numbers can represent a tree. We will call such a representation a *coding representation*.

On sequential storage media like magnetic tapes, the coding representation is convenient. Navigations along the relationship relation can be made by sorting or topologically sorting the file representing the relation on which the relationship relation is defined followed by a sequential read. However, it is not advisable on direct access storage media because maintenance of such code values is not so easy and the efficiency of navigations, which must be achieved by certain search operations, is worse than other representations.

(3) Pointer Field Representation

Association by pointer fields is the most commonly used and most efficient method for achieving navigations. Various pointer organizations are used according to the type of relationship relation to be represented.

If the relationship relation is a sequence or a quasisequence, it can be represented by a uni- or bidirectional pointer contained in the records representing the relation on which it is defined (uni- or bi-directional linear list).

If the relationship relation is neither a sequence nor a quasisequence, no single pointer can represent either or both directions of a relationship connecting two records. In fact, if the relationship relation is a tree, one origin record may have several destination records and hence a pointer array must be used to represent the forward direction of the relationship relation. (A single pointer can represent the backward direction.) If the relationship relation is a network, we need a pointer array to represent the forward direction and sometimes another pointer array to represent the backward direction.

However, in most cases a fixed number of pointers are used instead of pointer arrays. The following four propositions are basis of obtaining such representations.

P1. A tree can be converted into two quasisequences if a sequence can be defined on every set of destinations with a common origin.

P2. A hierarchy (a relationship relation defined between two relations and giving a one-to-many onto corresponding between these two

relations) can be converted into a quasisequence if a sequence can be defined on every set of destinations with a common origin.

P3. A multilevel hierarchy (a set of hierarchies whose skeleton becomes a tree) can be converted into a quasisequence if a sequence can be defined on every set of destinations with a common origin. A sequence obtained by connecting all conncted component of this quasisequence is sometimes called the *preorder*.

P4. Any relationship relation can be converted into an entity relation and two hierarchies.

A more precise description of these propositions together with their proofs are given in [Kobayashi 1981b]. Here we will only show conversions corresponding to P1, P3 and P4 respectively in Fig. 3, 4 and 5. P2 is a special case of P3.

As the result of convertion mentioned in P1, each of two quasisequences can be represented by a uni- or bi-directional pointer. This implies that a tree can be represented by two uni- or bi-directional pointers. Such a representation is sometimes called a hierarchical list.

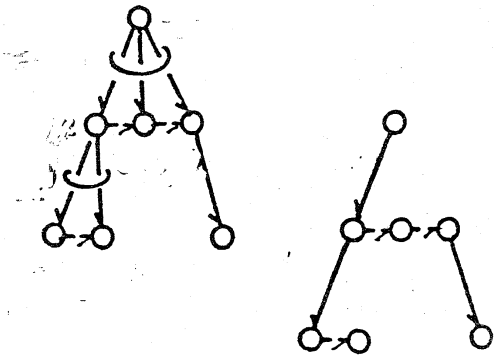

Fig.3 Conversion of a Tree into two Quasisequences.

If the tree to be represented is a hierarchy, a more simple representation by a single uni- or bi-directional pointer is possible as the consequence of proposition P2. In many database management systems developed according to CODASYL DBTG Proposal [CODASYL 1971], a ringed list is used to represent a hierarchy.

A multilevel hierarchy can also be represented by a uni- or bi-directional pointer as the consequence of proposition P3. this simple representation of multilevel hierarchies is used in most database management systems of hierarchical type [Tsichritzis 1976].

Proposition P4 is used to obtain a representation with a fixed number of pointers for a non-hierarchical trees and networks. Note that in this case a new (entity) relation representing the given relationship relation must be created.

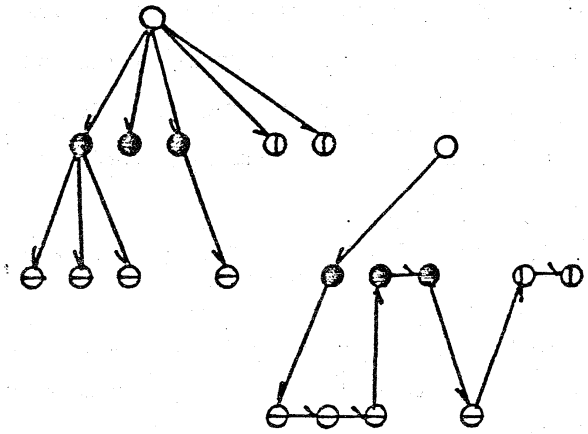

Fig.4 Conversion of a Multilevel Hiererchy into a Quasisequence.

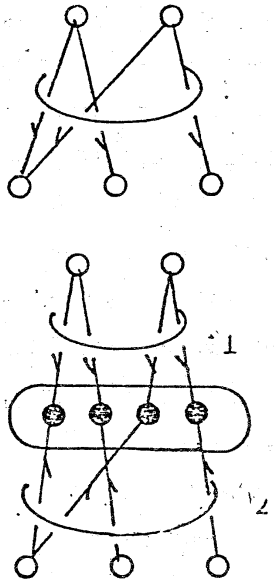

Fig.5 Conversion of an arbitrary Relationship Relation into a Relation and two Hierarchies.

Navigations along the paths specified by pointors are very efficiently achieved. The choice between uni-directional and bi-directional pointer organizations depends on the frequency of forward and backward navigations. If backward navigations are necessary but are requested not so frequently, a ringed list organization can be used. Note that we need navigations in both directions to update the relationship relation and the relations on which it is defined. In fact, any addition and deletion of records in these relations propagates update of several other records. Uni-directional pointers are insufficient to perform such updates efficiently. The update procedure is a little complicated but the update efficiency is not so poor if the file is static and bi-directional pointers are provided.

(4) Relationship File Representation

In addition to the above three we can represent relationship relations just like representing entity relations. This representation is mandatory for n-ary relationship relations when $n \geq 3$. Also if the relationship relation has some attributes other than its origin and destination attributes, we must create a file representing it. In both cases, n ($n \geq 2$) hierarchies can be defined each between this relation and one of relations on which it is defined, which can be represented by the pointer field representation to achieve a better navigation efficiency. If such a representation

of hierarchies is not employed, navigations must be achieved by a search with a search condition defined on two files or by two consecutive searches each with a search condition defined on a single file. In the latter case, the update efficiency is very good, since no propagated in updates are necessary.

LOCALIZATION

So far we have discussed a variety of representations of entity and relationship relations. For each relation a specific representation must be selected so that database operations frequently requested in user's applications are efficiently performed.

Physical read/write operations on these files are usually made with several records as a read/write unit. This unit is called a *block* or a *page*. A heap file as well as a sequential file is physically composed of a number of blocks sequentailly arranged in sequentail ot direct access storage devices. A stand-by buffer control technique can be used in an exhaustive (or sequential) search. A block may contain a partition in the partitioned sequential or tree-structured file organization. A bucket may correspond to a block in the direct file organization.

Block buffering can be regarded as a device that enables a quick access to a record r' which is the next (in some sense) to the record r currently fetched. In general, if a record r' is frequently fetched immediately after the record r is fetched, the time $t(r,r')$ required for fetching r' under the condition that r is currently fetched (in the workspace provided in the program area) must be as short as possible. Let $p(r,r')$ be the possibility of fetching r' immediately after fetching r. Here, the record r' is not necessarily in the same file to which r belongs. For the whole database DB, we have

$$\Sigma_{r' \in DB} p(r,r')=1$$

for all r, and

$$\Sigma_{r \in DB} p(r,r')=1.$$

for all r'. It is desirable to minimize

$$\Sigma_{r \in DB} \Sigma_{r' \in DB} p(r,r')t(r,r').$$

A general method to minimize the above is to divide a whole database into a number of pages of an appropriate size. Paging can be regarded as a device by which two records loginally near each other are

placed in two locations physicallu near each other. If we can form a probability matrix, whose (i,j)-coefficient is $p(r_i,r_j)$, we can divide the database into clusters each corresponding to a page. However, both estimating $p(r_i,r_j)$ values for all records in the database and clustering are too difficult for practical applications.

For the program, which is a set of instructions, paging or virtual storage techniques are very popular. However, in contrast to the program, which has a strong *locality*, the database has in most cases only a very weak locality. This makes the application of paging techniques to the database organization very difficult. Improper application of page buffering may even worsen the total efficiency.

In some applications, the extent of page clusters is obvious because of a specific characteristics of the data to be stored in the database. For example, in a cartographic database for some regional information system, page clustering can be made according to the geographical area subdivision on the map to be accommodated in it. In such cases, we can apply page buffering as a storage allocation algorithm for some parts of the database [Kobayashi 1980].

REFERENCES

[Astrahan 1975] M.M.Astrahan, and D.D.Chamberlin, Implementation of Structured English Query Languages, *Comm. ACM*, 18 (10), 580-588 (1975).

[CODASYL 1971] CODASYL Data Base Task Group, April 1971 Report, ACM (1971).

[Codd 1972] E.F.Codd, Relational Completeness of Data Base Sublanguage, in *Data Base Management*, Courant Computer Science Symposium, 6 (R. Rustin, ed.), pp. 65-98, Prentice-Hall, Englewood-Cliffs, New Jersey (1972).

[Held 1975] G.D.Held, M.R.Stonebraker, and E.Wong, INGRES: A Relational Data Base System, in *Proc. NCC*, pp. 409-416 (1975).

[King 1974] W.F.King, On the Selection of Indices for a File, RJ 1341, *IBM Research* (1974).

[Kobayashi 1976] I.Kobayashi, An Optimal Database Search Strategy for Retrievals on one File, *The Soken Kiyo*, 6 (2), 79-95 (1976).

[Kobayashi 1980] I.Kobayashi, Cartographic Databases, in *Pictorial Information Systems* (S.K.Chang, and K.S.Fu, eds.), Springer-Verlag,

(1980).

[Kobayashi 1981a] I.Kobayashi, On the Semantic Constraints and Normal Forms of Database Relations, *SANNO College of Management and Informatics*, TRCS-3 (1981). To appear in *International Journal of Policy and Information*, 6 (1) (1982).

[Kobayashi 1981b] I.Kobayashi, An Overview of Database Management Technology, *SANNO College of Management and Informatics*, TRCS-4, Rev. (1981). To appear in *Advances in Information Systems Science, Vol. 9* (J.T. Tou, ed.), Plenum, New York (1982).

[Kobayashi 1981c] I.Kobayashi, Evaluation of Queries based on the Extended Relational Calculi, *International Journal of Computer and Information Sciences*, 10 (2), 63-104 (1981).

[Knuth 1968] S.E.Knuth, *Soting and Searching, The Art of Computer Programming, Vol.3*, Addison-Wesley, Reading, Massachusetts (1968).

[Lum 1971] V.Y.Lum, and H.Ling, An Optimization Problem on the Selection of Secondary Keys, in *Proc. ACM Annual Conference*, pp. 349-356 (1971).

[Martin 1976] J.martin, *Principles of Data-Base Management*, Prentice-Hall, Englewood Cliffs, New Jersey (1976).

[Palermo 1970] F.P.Palermo, A Quantitative Approach to the Selection of Secondary Indexes, RJ 730, *IBM Research* (1970).

[Palermo 1972] F.P.Palermo, A Data Base Search Problem, in *Proc. 4th International Symposium on Computer and Information Sciences*, pp.67-101 (1972).

[Rothnie 1975] J.B.Rothnie, Evaluating Inter-entry Retrieval Expressions in a Relational Database Management System, in *Proc. NCC*, pp. 417-423 (1975).

[Senko 1973] M.E.Senko, E.B.Altman, M.M.Astrahan, and P.L.Fehder, Data Structures and Accessing in Data-base Systems, *IBM Systems Journal*, 12 (1), 30-93 (1973).

[Schkolnick 1975] M.Schkolnick, The Optimal Selection of Secondary Indices for Files, *Information Systems*, 1 (4), 141-146 (1975).

[Smith 1975] J.M.Smith, and P.Y.T.Chang, Optimization and Performance of Relational Algebra Database Interface, *Comm. ACM*, 18 (10), 568-588 (1975).

[Tsichritzis 1976] D.C.Tsichritzis, and F.H.Lochovsky, Hierarchical Database Management: A Survey, *Computing Surveys*, 8 (2), 105-123 (1976).

[Wiederhold 1977] G.Wiederhold, *Database Design*, Mc-Graw Hill, New York (1977).

[Wong 1976] E.Wong, and K.Youssefi, Decomposition: A Strategy for Query Processing, *ACM TODS*, L (3) 233-241 (1976).

[Yao 1976] S.B.Yao, Modeling and Performance Evaluation of Physical Data Base Structure, in *Proc. ACM Annual Conference*, pp.303-309 (1976)

[Yao 1979] S.B.Yao, Optimization of Query Evaluation Algorithms, *ACM TODS*, 4 (2), 133-155 (1979).